操作系统

进程：一个有独立功能的程序在一个数据集上一次动态执行的过程

OS **资源分配**的最小单位,分配了内存 I/O CPU

由**程序**(描述功能)、**数据集合**、**PCB** 组成(进程描述和控制信息--标志)

每个进程有独立内存，但因为进程间上下文切换耗费大，所以发明了线程

线程：**程序执行**和 OS **调度**的最小单位　并行

协程：是一种用户态的轻量级线程，调度的话完全由用户控制，创建时不用调用 OS 功能，多个协程在一个主线程上，是异步机制 为了并发

并发的关键是你有处理多个任务的能力，不一定要同时

并行的关键是你有同时处理多个任务的能力

差别：1.**定义**

进程是操作系统资源分配的最小单位，而线程是调度和程序执行的最小单位

2. 包含关系:进程(有多个线程) > 线程(进程种代码执行路线) > 协程

3. **共享**:进程相互独立，线程共享内存空间，堆等，但私有栈,寄存器(程序计数器)

4.**切换**:进程切换要保存当前 CPU 环境并设置新的，线程只是保存和设置寄存器

联系：通常只有一个 CPU 并只给一个进程，总线程数>CPU 数-并行，反之并发

并发是充分利用每一个核

线程的实现方法:

1. 线程池
2. 继承 Thread 类(编写简单，直接用 this 获取线程，但不能继承其他类了)
3. 实现 Runnable 接口(重写 run()方法，没返回值，不能抛出异常)
4. callable(重写 call()方法，有返回值，可以抛出异常)

3 和 4 适合多个线程资源共享

线程通信方法：1.**全局变量**(因为内存共有) **volatile**

2.**消息队列**(因为每个**线程有自己的消息队列**)

3.**事件**

线程 5 个状态：1.NEW 新建 2.RUNNABLE(其他线程 call 了 start 方法) 3.RUNNING runnable 的线程获得时间片 4.BLOCKED 放弃了 CPU<- sleep、锁被占用 5.DEAD

start()启动线程--处于就绪状态，之后 Thread 类调用 **run()使其运行**，run() 里面有线程内容，run 结束线程也终止

线程互斥：同时只能一个线程进入临界区(critical section)对资源进行读写操作

线程同步：多线程通过互斥量等机制控制线程之间执行顺序

比如一个线程对临界区写，另一个读，必须先写再读。不然结果不如预期

方法：1.互斥量 只有拥有互斥量对象的线程才能访问公共资源

2.临界区 只允许一个线程对共享资源访问
3.信号量 允许多个线程同时访问资源

线程安全：多线程访问时，用了加锁机制，当一个线程访问某个数据时进行保护，其他线程不能进行访问直到该线程读取完，不会出现数据不一致或者数据污染

线程不安全：不提供数据访问保护，有可能多个线程先后更改数据造成所得到的数据是脏数据，不符合预期

线程池：创建多个可执行的线程放到一个池里，需要的时候不用自己创建直接从池里获取，使用完不销毁线程而是把它放回池里，线程池会在空闲时间来创建和销毁线程，这样服务器在处理用户请求的时候就不会有创建和销毁线程的开销，可以控制最大并发数

应用场景：FixedThreadPool 用于负载比较重，而且负载比较稳定的场景，比如有套负载比较重的后台系统，每分钟要执行几百个复杂的 SQL, 因为负载稳定，一般来说，不会出现突然涌入大量请求，导致 100 个线程处理不过来，然后就直接无限制的排队，然后内存溢出

CachedThreadPool：用在负载很稳定的场景的话就浪费了。因为每天大部分时候可能就是负载很低的，用少量的线程就可以满足低负载，不会给系统太大压力；但每天如果有少量的高峰期，比如说中午或者是晚上，可能需要一下子搞几百个线程出来，那 CachedThreadPool 就可以满足这个场景，然后高峰期应付过去之后，线程如果处于空闲状态超过 1 分钟，就会自动被回收，这样就避免给系统带来比较大的负载

参数：

1. corePoolSize 核心线程数，指保留的线程池大小(不超过 maximumPoolSize 值时，线程池中最多有 corePoolSize 个线程工作)
2. maximumPoolSize 指的是线程池的最大大小(线程池中最大有 corePoolSize 个线程可运行)
3. keepAliveTime 指的是空闲线程结束的超时时间(当一个线程不工作时，过 keepAliveTime 长时间将停止该线程)
4. unit 是一个枚举，表示 keepAliveTime 的单位(有 NANOSECONDS, MICROSECONDS, MILLISECONDS, SECONDS, MINUTES, HOURS, DAYS，7 个可选值) 纳秒，微秒，毫秒，秒，分，小时，天
5. workQueue 表示存放任务的队列（存放需要被线程池执行的线程队列）
6. handler 拒绝策略(添加任务失败后如何处理该任务)

拒绝策略：

1. abortpolicy(线程池默认的拒绝策略，在任务不能再提交的时候，抛出异常，及时反馈程序运行状态。如果是比较关键的业务，推荐用这个拒绝策略，这样在系统不能承载更大的并发量的时候，能够及时的通过异常发现)
2. discardPolicy(丢弃任务，但是不抛出异常。如果线程队列满了，那么后面提交的任务都会被丢弃，而且是静默丢弃)
3. discardOldestPolicy(丢弃队列最前面的任务，然后重新提交被拒绝的任务)
4. CallerRunsPolicy(由调用线程处理这个任务,如果任务被拒绝了，就让调用线程（提交任务的线程）直接执行此任务)

有限阻塞队列 vs 无限的阻塞队列

有限阻塞队列(ArrayBlockingQueue)可能不能很好得满足性能，因为如果所有任务都没

办法执行完然后线程数又达到上限的话，新来的任务就会被用拒绝策略来处理了，这就需要调节线程数和 queue 的大小
无限阻塞队列(LinkedBlockingQueue)就是可以一直接受任务，但可能会耗尽系统资源

多线程：为了**同步完成多项任务，提高资源使用率**来提高系统效率(不是运行效率)，充分利用 CPU，更好地利用系统资源。
如果 OS 支持多个处理器，每个线程分配给一个处理器。
它最有价值的地方是我们不用知道使用了几个处理器。但可能会有**共享资源的问题**，所以要用**锁**
实例：生产者消费者问题，3 个地方卖 20 张票，用同步锁保证不会卖出同一张票
单核 CPU 支持多线程，通过给每个线程**分配时间片**

进程通信/同步方式：1.无名**管道**(有亲缘关系的进程间用，数据单向流动)
2.有名管道(没亲缘关系也可以用)
管道是在内核申请一块缓冲区用于读写
3.**共享内存**(映射一段物理内存到不同进程的虚拟内存) – 最快
最快因为**不用切换到内核态**，直接从**内存里读**
不会占用地址空间，因为内核为它增加了一段
注意：因为是**临界资源**，所以要用**信号量保证原子性**
4. 消息队列(一个队列，元素是数，进程可以访问这个队列)
5.**信号量**(计数器，作为一种锁的机制管理进程对资源的访问)Linux 里要用自旋转锁
6.**套接字**(用于**不同机器**间进程通信)

虚拟内存：使应用程序**被认为有连续完整的地址空间**，实际上是在多个**物理碎片**和**磁盘**(disk)上
ThreadLocal：**保证线程安全的方法，创建一个变量**后每个线程对其访问的时候都是**访问线程自己的变量** （Java 里）子线程能获得父线程 ThreadLocal

死锁：一组进程里每个进程都**有不释放的资源**，但又互相**申请**被其他进程**占的资源**而处于一种**永久等待状态**

死锁 4 个必要条件：

1. 互斥条件：资源由一个进程使用，不能共享
2. 请求与保持条件：已经得到资源申请新资源
3. 非剥夺条件：已经分配的资源不能从相应进程中剥夺
4. 循环等待条件：进程组成环路，每个进程都在**等相邻进程的资源**

最根本原因：没释放一个锁要去获得另一个锁

避免死锁：破坏四个条件之一，比如 2.为获取锁时间设超时时间、只用一把锁

银行家算法：允许进程**动态地申请资源**，系统在每次实施资源分配之前，**先计算**资源分配的**安全性**，若这次资源分配安全，就把资源分配给进程（即资源分配后，系统能**按某**

种顺序来为每个进程**分配资源**，使每个进程都可以顺利完成），否则**不分配**资源，让进程**等待**

sleep(): **让出 CPU** 暂停**执行**,**不释放锁**，**阻塞**线程，不考虑优先级，Tread 的静态方法

yield(): **让出 CPU** 暂停**执行**，**不释放锁**，**不阻塞线程**而是回到**就绪**状态，Tread 的静态方法

wait(): 进入**等待池(变成 blocked 状态)**，**释放锁**，只有别的线程调用 notify 才进入锁定池准备，Object 的方法，任何对象实例都能调用

join(): **等异步线程执行**完才能执行

block(): 线程正在等待获取锁 synchronized 会导致线程进入 blocked 状态

block() vs wait()：Blocked 是指线程正在等待获取锁，waiting 是指线程正在等别的线程调用 notify，之后线程可能会进入 runnable 状态，也可能再次获得锁变成 Blocked 状态

阻塞状态是线程因为某种原因放弃 CPU 使用权，暂停运行。直到线程进入就绪状态，才有机会转到运行状态。

阻塞的情况分三种：

等待阻塞：运行的线程执行 wait()方法，JVM 会把该线程放入等待池中。

同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则 JVM 会把该线程放入锁定池

其他阻塞：运行的线程执行 sleep()或 join()方法，或者发出了 I/O 请求，JVM 设置成阻塞。当 sleep()状态超时、join()等待线程终止或者超时、或者 I/O 处理完毕时，线程重新转入就绪状态

Shell 是一个应用程序，它连接了用户和 Linux 内核，让用户能够更加高效、安全、低成本地使用 Linux 内核

gdb：UNIX 和 UNIX-like 下的调试工具，用来运行程序，停在断点检查程序

缓存 cache 和缓冲 buffer：cache 是放被磁盘读出的　　buffer 是放写入磁盘的东西

cache 提高 **CPU 和内存**之间数据交换速度，把数据放在几级 cache 里，减少内存访问

buffer 提高**硬盘和内存**之间数据交换速度，把写操作集中进行来提高性能

文件描述符(file descriptor)是内存为了管理**被打开**的文件创建的**索引**(非负整数)

0 开始

IO 多路复用(同时处理多路 I/O)：有一堆文件描述符 -> 调用函数让 Kernel **监视**这些描述符，其中有描述符可以 I/O 读写操作再**返回**，返回后我们就知道哪些文件描述符可以来 I/O 操作

3 种 Linux **阻塞时(同步)I/O 机制**：

1. select：把**文件描述符**(最多 1024 个)通过**参数传给** select，select **拷贝**到 Kernel，线程**不安全**
2. poll：相似 select 只是文件描述符能超过 1024 个，线程**不安全**

3. epoll(Linux 独有)：**只操作有变化的描述符**，和 Kernel **共享**了些**内存**来放已经可读可写的文件描述符，**减少了拷贝到 Kenel 开销**

用**事件驱动来解决要遍历描述符才知道哪个可读可写**：进程只要等待在 epoll 上，epoll 代替进程去各个文件描述符上等待，哪个描述符可读可写就通知 epoll,epoll 记录下来然后唤醒进程

用户态(运行用户程序) **内核态**(运行 OS 程序)

区别：用户态比内核态**低级**，不能直接**访问 OS** 程序，占有的处理器**可被抢占**
用户态 -> 内核态 3 种情况:
1. (通常靠库函数)系统调用(不是进程切换) fork() –每个进程有两个栈 用户和内核栈
调用返回时再切换到用户态，如果队列里有更高优先级的进程就会进程切换
2.异常(发生异常会切换当前进程到处理此异常的内核)
3.设备中断(设备向 CPU 发出中断信号)，比如**硬盘读写完成 东西保存在 bios**
进程上下文切换就是 cpu 寄存器，内存栈，内存映射都发生切换。 用户态和内核态之间就会发生，进程间切换也是这种情况，进程切换是从一个进程切换的到另一个进程。

逻辑地址 -> 线性地址(虚拟) -> 通过 **MMU** 转化为物理地址

Linux 里**逻辑地址=线性地址** 因为**线性地址都是从 0 开始**
线性地址转物理地址：从进程中得到 pgdir 地址->用线性地址前 10 位找**索引**，中间 10 位找 page table 里 **page 位置** ->加上后 12 位得到**物理页**位置

Linux 3 种**内存模型**：

1. FLAT：访问**物理内存**时候，地址空间连续，不空洞，会浪费
2.Discontinuous： 地址空间不连续，有空洞
3.Sparse：连续的地址空间被**分段**，每段都支持**热插拔**

最佳置换算法 OPT 性能最好，但无法实现。先进先出 FIFO 简单，但性能差。
最近最久未使用 LRU 性能好，是最接近 OPT 性能的，但是需要专门的硬件支持，开销大。

时钟置换算法 CLOCK 是一种性能和开销均平衡的算法

CLOCK 算法思想：为每个页面设置一个**访问位**，再将内存中的页面都通过指针连成一个循环队列。当某个页被访问时，其访问位置 1.当需要淘汰一个页面时，只需检查页的访问位。是 **0 就换出**；是 **1 就把访问位改为 0**，继续检查**下一个**页面
若**第一轮扫描**中所有的页面**都是 1**，则将这些页面的访问位**都置为 0 后**，再进行**第二轮扫描**（第二轮扫描中一定会有访问位为 0 的页面，因此简单的 CLOCK 算法选择一个淘汰页面最多会经过两轮扫描）。

要用 Wait/waitpid:取得子进程运行状态，因为父进程和子进程是个**异步**过程(父进程不知道子进程什么时候结束)
孤儿进程：父进程退出，它的子进程还在运行->由 init 收集它状态 没有危害
僵尸进程：子进程退出，但父进程没用 wait 获得子进程状态，子进程的进程描述符仍

在系统中 ->多了导致没用可用进程号->不能产生新进程

静态库(扩展名.a/.lib)**compile 时直接整合到程序**，优点：compile 成功的文件可以**独立运行**，但函数库升级要重新 compile
动态库(.so/.dll) 文件需要用库函数才读取库函数(可执行文件无法独立运行)，优点：节省内存并**减少页面交换**，实现进程资源共享

OS 内存管理

a)最先适配：分配n个字节，查找并使用第一个可用的大于等于n个字节的空闲块。

b)最佳适配：查找并使用不小于n的最小空闲块。

c)最差适配：查找并使用不小于n的最大空闲块。

d)快速适配：为常用大小的空闲区建立单独维护的链表。例如：假设512B,1KB,4KB,10KB……为经常需要的大小的内存块，为这些内存块，建立单独的链表。进程申请内存时，通过该链表，直接使用这些空闲的内存块。

	优点	缺点
最先适配	简单 在高地址空间会留有较大块的空闲分区	在分配大块空闲块时，基本要遍历整个链表，所以时间会比较慢
最佳适配	可以最大程度的匹配进程需要的内存 避免大的空闲分区被拆分 恰好适配时，很好的减少外部碎片	释放分区慢 容易产生很多无用的小碎片
最差适配	避免出现小而多的空闲分区	释放分区较慢 容易破坏大的空闲分区
快速适配	可以快速的寻找指定大小的空闲分区	释放分区慢，临近空闲分区合并费时。

Java

创建对象的方法：1.new
2.通过 class 类的 newInstance class.newInstance()
3.通过 Constructor 的 newInstance
4.使用 clone 方法，需要实现 Cloneable 接口

异常处理机制

分为**抛出异常**和**捕捉异常** **异常分 Error 和 Exception(受检异常**：用 **try catch**
非受检异常：**运行**时错误，比如**/0)**

多线程异常处理：线程不允许抛出没捕获的 checked exception—线程要把自己 exception 在自己的 run 方法里用 try catch 处理掉，异常被 Thread.run()抛出后不能在程序捕获，只能靠 jvm 捕获。JVM 会调用 dispatchUncaughtException 方法来找异常处理器，再调用他的 uncaughtException()方法来处理异常的，并且是直接打印到控制台

Java 里是**值传递**，C 里是引用传递
值传递：调用函数时，将实际参数(地址)复制一份到函数中，函数对参数修改不会影响实际参数

引用传递：传实际参数的地址到函数，对参数修改会影响实际参数

空指针发生的情况：
1.访问或修改 null 对象的字段。
2.如果一个数组为 null，用属性 length 获得其长度时，访问或修改其中某个元素
3.在需要抛出一个异常对象，而该对象为 null 时
4. 先判断是不是 null 再判断 equal(“”) 不然会报错 NullPointer

实现 **Serializable** 接口实现序列化(程序不运行仍保存其信息)—把对象以字节序列的形式保存
反序列化：通过字节序列得到原对象。用于系统间通信

泛型：通过参数化类型来实现在同一份代码上操作多种数据类型

类型擦除：List<Object>和 List<String> compile 后都会变成 List
泛型擦除：

- 若泛型类型没有指定具体类型，用Object作为原始类型；
- 若有限定类型< T exnteds XClass >，使用XClass作为原始类型；
- 若有多个限定< T exnteds XClass1 & XClass2 >，使用第一个边界类型XClass1作为原始类型；

4.0-3.6=0.400000001 因为 **2 进制的小数无法精确地表达 10 进制小数**

接口与抽象类

都用 abstract 声明，有抽象方法的类一定要声明成抽象类
抽象类不能被实例化(类名 a = new 类名())，只能被继承
区别：

抽象类 (is-A 关系)	接口 (like-A 关系)
有方法实现和构造器	Java8 之后有方法实现，无构造器
子类不是抽象类的话要提供所有方法实现	子类都要提供所有方法实现
方法可以有 public protected	方法只能 public

一个类能实现多个接口但不能继承多个类

集合类

1.来自 **Collection**(List,Set,Queue)
List：按位置存储数据的对象，有序 LinkedList
Queue：PriorityQueue 为元素提供优先级
Set：和 List 区别是不允许重复元素
除了 Map 类都 iterable
2.来自 **Map**(TreeMap--基于红黑树实现)
查找效率是 O(1), HashMap, LinkedHashMap(保证顺序)

重写和重载 Override 和 Overload

重写：子类实现父类方法声明一样的方法
子类方法访问权限大于父类(父 protected，子 public)，
子类返回类型和抛出异常类型和父类一样或为其子类型

(父 List<Integer>子 ArrayList<Integer>)
static(静态)方法只能被继承不能被重写
重载：同一个类中两个方法名字一样，但是参数类型、个数、顺序至少有一个不同

面向对象(OOP)是一种思想 三大特性

1.封装：将类的某些信息只能类内部访问，public 封装性最差，this
2.继承：子类拥有父类所有属性和方法, 除了 private
static(静态)方法只能被继承不能被重写,，final 不被继承和重写
3.多态：对象的多种形态--必要条件：继承，重写，向上转型
实现多态：继承和接口
1.引用多态:父类引用指向本类和子类对象：Animal obj1 = new Dog();子类
2.方法多态:创建子类对象，调用子类重写的方法或继承的方法

拷贝

clone(**浅拷贝**)拷贝基本类型值 深拷贝要实现 cloneable 接口并重写 clone 方法
引用拷贝：Object obj = new Object() Person cur = obj

String, StringBuffer, StringBuilder

3 者都是用 char 数组存字符串
String 是不可变类，对 String 进行改变会生成新的 String 对象，然后指针指向新生成的 String，浪费内存
StringBuffer 和 StringBuilder 的对象被修改不会产生新的对象，所以字符串经常改变的话用它们
StringBuffer 线程安全因为所有公开**方法**都是用 **synchronized 修饰**的但因为要加锁所以性能差 速度慢
每次获取 toString()都会直接用缓存区 toStringCache()值来构造一个字符串
StringBuilder 不是线程安全的，不能被同步访问，但速度快
需要复制一个字符串数组再构造一个字符串
一般用 StringBuilder，除非要考虑线程安全

JVM

JVM 内存组成：1.计数器(当前线程执行的字节码的行号指示器，(PC)存指向下条命令的地址，切换线程后知道要从哪开始执行，线程私有) 2.虚拟机栈(方法执行的模型，线程私有) 3.本地方法栈(线程私有) 4.堆(共享) 5.方法区(一块线程共享内存)

JVM 内存模型决定一个线程对共享变量的写入什么时候对另一个线程可见

把线程间共享变量放在主内存，每个线程都有一个私有的本地内存
JVM 栈(放引用)：
1. **存**局部变量和方法调用，FIFO 数据结构，
2.**空间**比 Heap 小，创建和释放存储空间快，因为只要移动栈顶指针
3.栈内存属于单个线程因为线程**不共享栈**
JVM 堆 – 被分成新生代和老年代
1. **存**储实例化对象和数组

2. **分配内存花的时间久**
3. 内存中所有对象**对线程可**见

垃圾回收---GC　垃圾：　无引用的对象

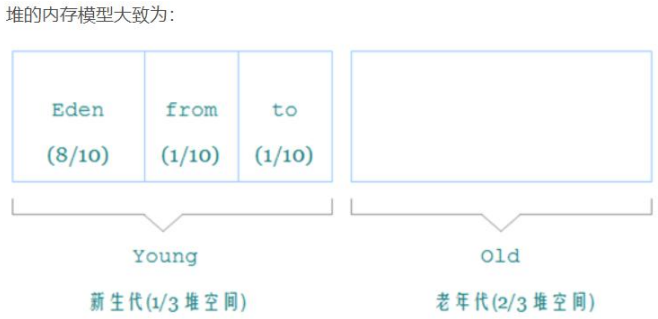
JVM 提供的在空闲时间不定时**回收无对象引用的对象占据的内存空间的机制**
通知 GC 回收：1.System.gc　2.把不用的对象赋值 null
引用类型：存的值是另一块内存的起始地址
强引用：只要引用存在就不回收 Object obj = new Object();
软引用：系统**内存不够**时回收，描述**可能有用**的对象
弱引用：**GC 工作时**就被回收
虚引用：**无法取得对象实例**，只是希望被回收时**收到系统通知**
解决循环引用(只用强引用引起，导致对象永远不能被释放)：如果 A 强引用 B，那 B 引用 A 时就用弱引用，判断是否为无用对象时仅考虑强引用计数是否为 0，不用关心弱引用数

判断垃圾算法：1.**引用计数算法**(引用->计数器+1，引用释放-1，**0 时清除**)

2.**根搜索法/可达性分析法**(从 **GC Root(虚拟机栈、静态变量、线程)**开始**往下**搜索，**清除没引用链的对象**)

回收算法：1.标记清除法(**标出所有**要回收的对象统一回收)--效率低

2.标记整理法(**标出**要回收的对象，让存活的向一端**移动**，直接清边界外的)-无
3.复制法(把内存**分成两块**，先用一块，把**存活的对象放另一块**)-占内存
4.分代收集法(把内存分成**新生代**和**老年代**，分别用**复制法**和**标记整理法**)
新生代分 3 个区域 Eden，from Survivor，To Survivor—这样分时为了更好地管理堆内存里的对象，包括内存的分配和回收



不分代的话得所有 root(一组活跃的引用)要扫一遍然后要知道是不是在引用链上还要扫一遍
对于新生代，对象存活期短，所以用复制法，只用关心哪些要被复制，只用标记和复制很少的存活对象，不用遍历整个堆，因为大部分是要丢弃的，缺点是浪费一半内存
对于老年代对象特点用标记-清理法，这部分如果用复制法的话没有额外空间担保(因为新生代放不下会放到老年代)，而且对象存活率高，复制的开销大
如果不分代的话，因为老年期的对象长期存活，总的 gc 频率和分代以后的年轻代 gc 频率差不多，但每次都要从 gc roots 完整来堆遍历，大大增加了开销

内存泄漏：非必要的对象引用**没清除**(数组里**添加对象后不处理**)，**未释放的资源**
内存泄漏的情况：

1.静态集合类如 hashmap、Linkedlist，如果这些容器是静态的，它们的生命周期与程序一致，容器的对象在程序结束前不能被释放。
2.改变对象哈希值导致 hashset 中无法删除当前对象
3.一个变量的定义的作用范围大于使用范围
4.内部类持有外部类
5.各种连接，比如数据库连接，没用 close 方法就不会去回收了

智能指针：一种抽象的**数据类型**，会**跟踪**指向它们的**对象进行内存管理**，通过使资源自动分配来**防止内存泄漏**。指向对象的指针被破坏时(超出范围)，对象**也会被破坏**

反射：Java compile 后会生成.class 文件，反射就是通过**字节码文件**找到类提供**类运行时的信息**。借助 **field**(可以用 get 和 set 得到和修改 field 对象) **method，constructor，class** 这 4 个类。　　缺点：性能消耗大

反射实现原理：在 JVM 层面，java 的**对象引用**不仅要可以**直接**或间接地得到**对象类型**，更应该可以根**据索引**能得到对象类型

JVM 加载.class 文件：类加载器将类的.class 文件中的二进制数据读入内存，将其放在运行时数据区的方法区内，然后再 heap 创建 java.lang.Class 对象用来封装类在方法区的数据结构

双亲委派：类加载器**加载.class 文件**时，先**递归**地把这个任务委托给上级类加载器，上级类加载器没加载，自己才加载

作用：**防止重复加载**一个.class 文件，保证核心.class 文件不被篡改

JVM 里 new 对象时, 堆会发生抢占吗?怎么去设计 JVM 的堆的线程安全

会，假设 JVM 虚拟机上，堆内存都是规整的。堆内存被一个指针分为两部分，左边塞满了对象，右边是没使用的区域。每一次 new 对象，指针就会向右移动一个对象 size 的距离。如果我们多线程执行 new 对象的方法，一个线程在给一个对象分配内存，指针还没有来的及修改，其它线程给另一个对象分配内存，而且这个线程还是引用之前的指针指向，这样就会出现抢占，就是**指针碰撞**

解决：在 JVM 新生代开辟一块 TLAB(线程本地分配缓存区)，也就是一块线程私有的内存分配区域。然后用这块区域放小对象，因为 Java 里有很多小对象而且小对象都是用完就被垃圾收集的，也不会被线程间去共享，而且因为线程私有所以对象分配的时候不用锁住整个堆，不存在竞争的情况，直接在自己缓冲区分配就行

TLAB **缺点**：

但 TLAB 只是让每个线程有私有的分配指针，底下存对象的内存空间还是给所有线程访问的，只是其它线程无法在这个区域分配而已 +大小固定

线程同步的实现/机制

1.Synchronized(独占锁, 可以修饰方法和代码块)

原子性：两个线程无法同一时间操作数据(要连续完成) + 线程访问的同步有序性

底层原理：线程 1 要加锁，执行 monitorenter 指令->看计数器 -> 0 就**获取锁并计数器改为 1** -> 线程 2 想加锁执行了 monitorenter 指令发现计数器是 1 就阻塞->线程 1 执行完释放锁，执行 monitorexit 将计数器置为 0->线程 2 进入

2. **Lock** 保证多个线程都是读或写时候就能一起进入 (CAS(保证了原子性—串行)和 volatile 实现) 保证一个线程有资源，其他 **spin**

ReentrantLock(重入锁)实现**线程安全**: 底层通过 **CAS 操作和 AQS 队列**去维护 state 变量的状态

流程：先通过 CAS 操作尝试修改 state 状态获取锁，如果获取失败就判断当前占用锁的是不是自身，如果是的话就进行重入，如果不是就进入 AQS 队列等待

3. **volatile**(线程同步的轻量级体现，但只能用于**变量**)

可见性：变量被 volatile 修饰后一个线程修改这个变量值其他线程可见

有序性：代码执行顺序就是语句顺序(不会发生指令重排(为了效率))

底层原理：修饰后在总线开启 MESI **缓存协议**和 CPU **总线的监听** -> 这个变量值**修改就会**传到总线 -> 总线**告诉其他**在使用这个变量的线程 -> 其他线程**获取新**的值

4.**wait()/notify()**

volatile 比 Synchronized 性能好

Lock 有 Synchronized 所有功能，但需要手动加锁/解锁(因为可以手动控制，所以再复杂的并发场景用得上), Synchronized 会加锁解锁，操作简单，用于一般并发场景

ReentrantLock 和 synchronized 的区别

1. 从底层实现上来说，synchronized 是JVM层面的锁，是Java关键字，可以修饰方法和代码，它是通过monitor对象来完成，对象只有在同步块或同步方法中才能调用wait/notify方法

ReentrantLock 是从jdk1.5（java.util.concurrent.locks.Lock）提供的API层面的锁

synchronized 的实现涉及到锁的升级 (具体为无锁、偏向锁、自旋锁、向OS申请重量级锁)

ReentrantLock实现是通过利用CAS（CompareAndSwap）自旋机制保证线程操作的原子性和volatile保证数据可见性来实现锁

2.能不能手动释放：

synchronized不用用户释放锁，它的代码执行完系统会自动让线程释放锁

ReentrantLock要手动释放锁，没释放可能会导致死锁，在final语句块里解锁

同一个线程在外层方法获取锁的时候，进入内层方法会自动获取锁。好处：避免死锁

3.能中断：

synchronized 是不可中断类型的锁(除非加锁的代码中出现异常或正常执行完成)

ReentrantLock 可以中断，(可以通过 trylock(long timeout,TimeUnit unit)设置超时方法或者将 lockInterruptibly()放到代码块中，调用 interrupt 方法进行中断）

4.公平锁？

synchronized 是非公平锁

ReentrantLock 可以是公平锁也可以是非公平锁(通过构造方法 new ReentrantLock 时传入 boolean 值进行选择，为空默认 false 非公平锁，true 为公平锁)

5.能不能绑定 condition：

synchronized 不能绑定

ReentrantLock 可以绑定(通过绑定 Condition 然后结合 await()/singal()方法实现线程的精确唤醒，而不是像 synchronized 通过 Object 类的 wait()/notify()/notifyAll()方法要么随机唤醒一个线程要么唤醒全部线程)

6.锁的对象

synchronzied 锁的是对象，(锁保存在对象头里面，根据对象头数据来标识是否有线程获

得锁/争抢锁)
ReentrantLock 锁的是线程，(根据进入的线程和 state 标识锁的获得和争抢)

AQS（Abstract Queued Synchronizer）

AQS 是一个用来构建锁和同步器的框架，可以构建比如 ReentrantLock，信号量，ReentrantReadWriteLock

核心思想：如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，同时把共享资源设置为锁定状态。

如果被请求的共享资源被占用，那就要一套线程阻塞等待和被唤醒时锁分配的机制，这个机制 AQS 是用 CLH 队列锁实现的，就是把暂时获取不到锁的线程加入到队列里。然后 AQS 使用一个 voliate int 成员变量来表示同步状态，它是通过内置的 FIFO 队列对获取的线程排队

AQS 用 CAS 对这个同步状态进行原子操作来实现对它的值的修改。

AQS 两种**资源获取方式**：独占式(只有一个线程有锁，又根据是否按**队列的顺序**分为公平锁和非公平锁，如 ReentrantLock)和共享式(多个线程同时获取锁访问资源，如 Semaphore/CountDownLatch，Semaphore、CountDownLatCh、 CyclicBarrier)
ReentrantReadWriteLock 可以看成是组合式，允许多个线程同时对某一资源进行读

AQS **底层**：用了模板方法模式，自定义同步器了，然后在实现的时候只要实现共享资源 state 的获取与释放方式就可以了(至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS 已经在上层已经帮我们实现好了)

CLH 同步队列是怎么实现非公平和公平的？--公平锁和非公平锁

CLH(FIFO 的双向链表)用于等待资源释放的队列，AQS 用它来管理同步状态

公平锁和非公平锁在于 hasQueuedPredecessors()方法，如果头节点不是尾节点，第一个节点不为空，而且当前节点是头节点就返回 true

线程在 doAcquire 方法里获取锁的时候会先加入到同步队列，之后再根据情况再陷入阻塞

阻塞后的节点一段时间后醒来，这个时候来了更多的新线程来抢锁，这些新线程还没有被加到同步队列里去，也就是还在 tryAcquire 方法里获取锁

在**公平锁**情况下，这些新线程会发现同步队列里有节点在等待，然后这些新线程就不能获取到锁，就去排队了

在**非公平锁**下，这些新线程会跟排队苏醒的线程抢锁，失败的线程就去同步队列里排队。所以这个**公不公平针对的是**苏醒线程和还没加到同步队列的线程，

然后根据 AQS 节点唤醒机制和同步队列的 FIFO 性质，那些已经在同步队列里阻塞的线程，它们内部其实是公平的，因为它们是会按顺序被唤醒的

Java 锁的类型

1.乐观锁：拿数据时认为别人不会修改，所以不会上锁，但是在更新时候会
适用于读操作多时，实现方式有 compare and swap(CAS)算法
CAS 是线程安全的，因为他有原子性(全执行成功 or 全失败)

悲观锁：每次拿数据都认为别人会改，所以每次都上锁
适应于写操作多的场景，因为加锁保证写时数据正确

synchronized 和 lock 都是悲观锁

2.独享锁(一次只能一个线程持有) 共享锁(可以被多个线程持有)

实现：互斥锁和读写锁

3.公平锁：线程按申请锁的顺序获取锁，非公平锁相反(都是双向链表)

非公平锁优点：吞吐量比公平锁大

4.可重入锁：同一个线程在外层方法获取锁的时候，进入内层方法会自动获取锁。

好处：避免死锁

Java IO

针对被调用者：

同步：发起调用 ->被调用者处理完请求再返回(调用者等待)

异步：调用 -> 被调用者立马回应收到(没返回结果)->调用者处理其他请求->被调用者依靠回调、事件等机制返回结果

针对调用者：

阻塞：发起请求，调用者一直等请求结果返回

非阻塞：发起请求，调用者不用一直等，可以去干别的事情

Java 对 OS 的 IO 模型的封装：

1.BIO(Blocking IO):同步阻塞模型，数据读写阻塞在一个线程内完成，应用于连接少，低负载和并发，JDK1.4 之前

2.NIO(Non-Blocking IO):同步非阻塞模型，读写数据到缓冲区，应用于连接多且短，高负载和并发，比如聊天服务器，JDK1.4 开始有

3.AIO(Asynchronous IO)：异步非阻塞模型，基于事件和回调实现，也就是操作后直接返回，不会堵塞在那，当后台处理完成，OS 通知响应线程进行后序操作, 应用于连接多且长，比如相册服务器，JDK7 开始有

Object 类的方法：equals() hashCode() wait()

equals 重写场景：比较对象内容 String，Integer -> 使 equals 和= =不同

hashcode 重写场景：不希望造成多个对象 hash 值相同，值不被覆盖

相等的对象必须有相等的哈希码，两个对象 hashCode 相同他们不一定相同

Hashmap 和 Hashtable 区别

底层都是哈希算法，接口一样

HashMap：区别：1.线程不安全(多线程下 put 会形成环导致死循环)因为 1.7 里扩容会造成数据丢失 1.8 会有数据覆盖 2.可以存 null:null 3.继承 AbstractMap 类

put 原理：先对 key 用 hashcode 方法，然后用 hashcode 的结果找到 bucket 位置来存键值对，也就是作为了 Map.Entry，如果 hashcode 一样就用 equals 比较，因为 hashcode 一样不一定就相同，还一样就替换

get 原理：对 key 用 hashcode 后得到桶位置，之后用 equals 找链表里的节点

作为 key：String，Integer 这样的 wrapper 类，因为他们是不可变—线程安全，也是 final 的而且重写了 equals 和 hashcode 方法

1.7：版本当哈希冲突严重时，桶上的链表会很长，查询很慢 O(n)

1.8：链表长度小于 8，用链表 因为查询快

链表长度 8+数组长度大于 64 转化为红黑树--泊松分布统计得出

查询为 O(logn) 不用 AVL 树因为插入太慢，经常会调整树的结构

扩容：初始大小(固定 2 的倍数)=16 扩容因子=0.75 扩容一倍 当前 12 容量 16 再扩容。0.75 是提高空间利用率和减少查询成本的折中，主要是泊松分布，0.75 的话碰撞最少大小。

扩容机制：单线程调用 rehash 把链表遍历，把元素每次用 transfer 方法放到新的链表头，链表元素次序会反过来因为每次都是从头插入来避免 O(n)去遍历尾部

多线程下 transfer 会导致**死锁**，比如 A.next=B, B.next=A—用 ConcurrentHashMap

为什么 HashMap 不用平衡树(AVL)--解决二叉树退化成链表的情况

因为**平衡树(AVL 树)**追求绝对平衡(search 时候更好)，条件比较苛刻(左右子树高度差不能超过 1)，实现起来也比较麻烦，每次插入和删除节点都要判断，然后之后需要旋转的次数也不能预知。总之就是实现的代码量会特别大，而且会增加额外开销，所以用红黑树

红黑树用颜色标识高度是大致平衡的，在和平衡二叉树的时间复杂度相差不大的情况下，能保证每次插入最多只要三次旋转就能平衡，实现起来也更简单

他们插入节点 rebalance 都要 O(1)，但删除的话 AVL 树就要 O(logn) 因为要维护从被删除节点到根节点 root 这条路径上所有节点的平衡

红黑树的性质：

1. 节点是红色或黑色 2. 根节点是黑色 3. 每个叶子节点都是黑色的空节点（NIL 节点）4. 每个红色节点的两个子节点都是黑色(从每个叶子到根的所有路径上不能有两个连续的红色节点)5. 从任一节点到它每个叶子的所有路径都有相同数目的黑色节点

红黑树的性质导致了**大致平衡**：

从根到叶子的最长的可能路径不会大于最短的可能路径的两倍长。因为操作比如插入、删除和查找某个值的最坏情况时间都要求和树的高度成比例，这样就允许红黑树在最坏情况下都是高效的，(这个是不同于普通的二叉查找树)

HashTable：区别：线程安全(每个方法已有 synchronize 方法，所以多并发时候用)，key 或 value 不能为 null，继承 Dictionary 类(废弃了)，用 hashcode 取余计算 hash 值

HashSet 实现了 Set 接口，是 HashMap 的一个实例，只存 key 不存 value，因此重写了 equals 和 hashCode 来判断 Key 存不存在(先用 hashcode 和其他对象比较结果，一样再用 equals，返回 true 说明有一样的存在)，不存在才插入，所以 key **不重复**

ConcurrentHashMap

1.7(两个静态内部类)(底层是分段数组和链表)用 HashEntry(封装映射表的键值对)和 Segment(充当锁)守护若干个桶(每次在每个链表头插入节点，所以节点顺序和插入顺序相反)每个线程只能访问不同的段，提高了并发率

线程安全, 因为它用了分段锁，把一个 HashMap 切割成 Segment 数组，然后 Segment 可以看成一个 HashMap，然后 Segment 继承了 ReentrantLock，在操作的时候会给 Segment 一个对象锁，想要修改 HashEntry 必须获得对应的 Segment 的锁 但是容易冲突让链表太长，这样查询就慢了

1.8(底层和 HashMap 一样链表长度大于 8 变成红黑树)放弃了 segment，在节点上用

CAS+Synchonized(锁住红黑树的 root)来保证**并发的安全**，只要 hash 不冲突就不会有并发问题。然后是用 volatile 关键字来记录元素个数，这样变量值变了的话其他线程也能看见就不会脏读了，提升了可见性

优化了什么/为什么用：

HashMap 用 synchronized 修饰，对象整体会被锁住了

HashTable 大小增加到一定的时候，性能会急剧下降，因为迭代时候要被锁很久

ArrayList、LinkedList

1.Array 能**包含基本类型和对象类型 + 大小固定**

2.ArrayList 只能包含**对象类型+大小动态变化** 访问效率高, 插入和删除可能要移动整个 list。**初始大小=10 每次 1.5 倍扩容**

Add 底层：add(元素)会加在最后，不会触发底层的数组的复制，add(元素,index)会触发数组复制，最坏时间复杂度是 O(n)。new ArrayList 的时候是一个空的 object 数组，大小=0，第一次加元素，大小变成 10，每次 add 都会去检查数组空间够不够，不够就用 grow 方法扩容 1.5 倍，扩容完用 System.arraycopy 拷贝数组

Remove 底层：删除一个元素后把后面的元素往前移一个 index 并且把最后一个 index 设为 null，底层移动是用 System.arraycopy(elementData, index + 1, elementData, index, numMoved);把后面的元素到前一位

3.LinkedList 增删节点效率高，只是修改了引用地址所以是 O(1)，但链表定位到要增删的地方是 O(n)，但写入是 O(1)，Arraylist 是 O(n)。所以 Arraylist 擅长读取，链表擅长写入　　双向链表　没有扩容机制

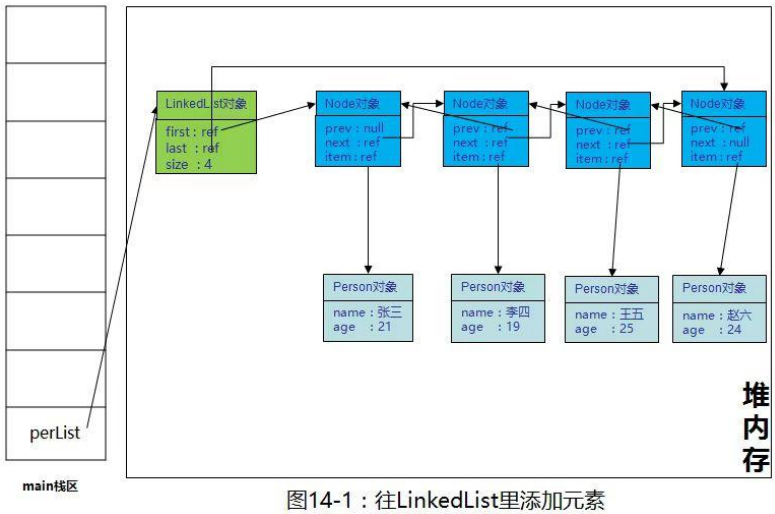


图14-1：往LinkedList里添加元素

Add 底层：

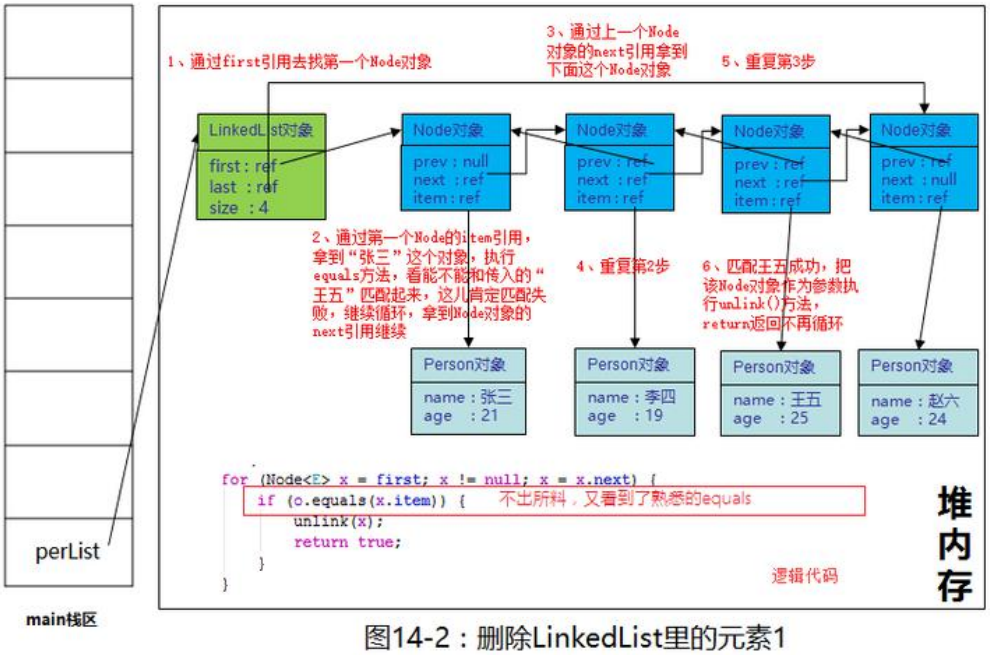


图14-2：删除LinkedList里的元素1

Remove 底层：

ArrayList(无原子性)和 LinkedList **变安全**：进行读操作时获取读锁，进行增删改操作时获取写锁(用读写锁类 ReentrantReadWriteLock)

4.Vector 也是用数组方式存储，但加了 synchronized 修饰，所以线程安全，但是性能比 ArrayList 差

三大设计模式：创建型(单例模式)，结构型，行为型

单例模式：一个类只有一个实例，并提供一个它的全局访问点(性能高)

懒汉式单例模式：用静态内部实现，使用时才会创建实例对象

```
//饿汉模式
public class Singleton {
    private static Singleton instance = new Singleton();
    private Singleton() {

    }
    public static Singleton getInstance() {
        return instance;
    }
}
//懒汉模式 线程不安全
public class Singleton {
    private static Singleton instance;
    private Singleton() {
        if(instance == null) {
            instance = new Singleton();
        }
    }
    public static Singleton getInstance() {
        return instance;
    }
}
//懒汉模式 线程安全
public class Singleton {
    private static Singleton instance;
    private Singleton() {
    }
    public static synchronized Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

保证一个类只有一个实例。实现方法：实例存在直接返回，不存在才创建实例

工厂模式(创建型)：创建接口，让子类决定实现哪个类

代理模式(结构型)：给其他对象创建代理来控制对一个对象的访问
观察者模式(行为型)：改变一个对象会同时改变其他对象，用于消息队列
生产者消费者：
1. 当队列为空时，消费者线程阻塞，否则唤醒消费者线程
2. 当队列为满时，生产者线程阻塞，否则唤醒生成者线程
因为消费者只从队列里面拿数据，用 take 方法
而生产者只放东西到队列，用 put 方法，这两个方法是独立的

java SE 个人计算机应用 eclipse 里用
java EE 是 java 企业版，用于服务端应用，提供 web 服务，组建模型和管理和通信 API，可以用来实现企业级的面向服务体系结构
java ME—微型版，用于移动产品和车载产品，给设备提供灵活的界面和健壮的安全模式。jvm 把.java 文件编译成字节码文件.class，之后不同的 jvm 都可以解析.class 文件来在不同 OS 上执行